Terminologies

IO intensive task – where the processor regularly wait for Input/Output – like reading from disk, writing to a disk, network REST svc calls, interacting with db

Anything which is slower than main memory, where reading is slow like network calls, db calls

Human thread pool is called call center

Facts

Creating threads is costly affair , so in real time u don’t create thread, you should always submit a task to thread pool like executor service thread pool

Advantages

1. If Thread dies suddenly we don’t need to worry, another thread from pool will be assigned to work on our submitted task
2. We don’t need to create thread object and we don’t need to call start method

Executor service

A class called

|  |  |
| --- | --- |
| *ExecutorService* es = Executors.*newFixedThreadPool*(20); | //worst as some would have blocked for IO  And we cant increase number of threads based on load |
| Executors.newCachedThreadPool() | Ok—threads are created on demand/workload – but this keeps on increases if more tasks came  Best is Threads will be shutted down automatically if they are not used |
| Executors.*newWorkStealingPool*(3); | Here among 3 threads if any thread is ideal, that thread will steal that work from other threads queue |
| *ScheduledExecutorService* scheduledes = Executors.*newScheduledThreadPool*(2); |  |
| ThreadPoolExecutor | Executes each submitted task using a pooled worker thread |
|  |  |
|  |  |
|  |  |

Key interfaces

Executor, ExecutorService, ScheduledExecutorService, CompletionService,

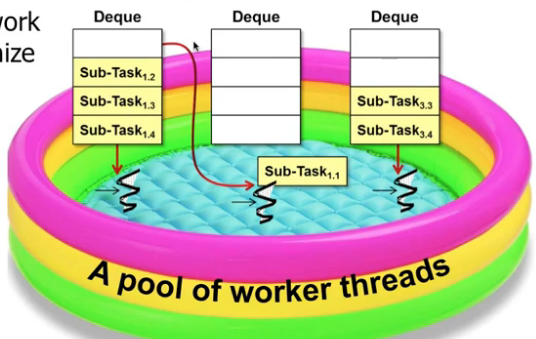
Tasks

These tasks represents the tasks whose instances are executed by another thread

|  |  |  |
| --- | --- | --- |
| Runnable | Callable | Future (I) & impl class is FutureTask(c)  Future (represents result of an async task, like result of callable) |
| Void run() | Object call() | boolean cancel(boolean *mayInterruptIfRunning*);  boolean isCancelled();  boolean isDone();  *V* get() – this is a blocking call & returns the result after some thread executed that task  *V* get(long *timeout*, *TimeUnit unit*)  Callable<String> task = () -> searcher.search(target);  Future<String> future = executor.submit(task); |

dfg

Stealing pool



Here if u see 2nd thread has completed all subtasks, as this is ideal, this will steal another sub tasks which belong to other thread

Here deque- is called double ended queue, means from both sides we can pick the tasks,

Note:- while using this steal pool, its our responsibility to split or fork a main task into multiple subtasks, else other idle threads cant pick those subtasks,

If there is only single task, that task will be owned by a thread, so other threads cant come and pick partially, hence better to split as sub tasks

Note:-

1. so always use thread pools , instead of dynamically creating or spawning a thread per client incurs excessive processing overhead

Fork join pool

Here forking means – splitting into smaller tasks (until u cant split further anymore)

This is based on divide and conquer, it says

If

Problem is small solve it directly

Else

Split into independent parts

Run all sub tasks parallelly (Fork new sub tasks to solve each part)

Join all those sub-tasks

1. split into individual smaller sub tasks (if u split a big task into smaller then only other ideal threads can take this smaller sub tasks,

else only that dedicated thread need to run this big task meantime if other threads are ideal they cant take as it is not a sub task)

1. all these sub tasks will be run in parallel by n threads of same core or diff core processor
2. merge the sub tasks results using join() 🡪 waiting for them to complete or

This is similar to executor service with below diff

Similarities

Here also we will submit the tasks to pool like in executor service

Difference

1. Here work stealing will be there
2. If 20 threads are there for 10 tasks, then each thread will take 1 task, now remaining 10 will be ideal (in fork join pool these will not be ideal if we define those tasks as even more sub tasks), whereas in fork join pool, we should fork means we should divide into even small parallel tasks (not IO operations like db , REST service calls as we can’t split these tasks), now when we split each task into even small tasks

Then remaining ideal threads will tasks these small tasks

When to use this

1. ForkJoinPool may also be appropriate for use with event-style tasks that are never joined.
2. We should not use this IO operations (like db calls, network REST calls)